Apunte 20 - Seguridad en el Backend

Introducción

Siendo la Seguridad un tema tan amplio, este apunte se limitará a discutir los aspectos básicos de la seguridad de una aplicación de Backend que expone diferentes endpoints. Esto significa que se asumirá que el cliente es capaz de realizar los procesos necesarios de autenticación y autorización sin ninguna interfaz gráfica.

Conceptos básicos

En esta sección se explican, brevemente, algunos conceptos que serán de utilidad para comprender el resto del apunte.

Autenticación

Es el proceso por el cuál un usuario (en este caso del backend) demuestra ser quien dice ser. Por ejemplo, para acceder a la información del sistema de Autogestión de la Universidad, esta demostración la hace el alumno proveyendo un nombre de usuario y una contraseña.

Autorización

Autorización permite verificar que un usuario autenticado pueda acceder únicamente a los recursos sobre los que tiene permiso. Volviendo al ejemplo de Autogestión, que el usuario 150000 se haya *autenticado* correctamente no significa que pueda acceder a las notas del usuario 234000, ni que pueda modificar las suyas.

Codificación

Es el proceso de convertir datos en un formato específico. Por ejemplo, se tiene una secuencia de bytes como la siguiente:

```
00 02 3f 5a 48 97 21
```

Y se desea transmitirla como texto (por ejemplo para transmitirla en una petición HTTP), entonces esa cadena puede ser codificada como *Base64* (una codificación frecuentemente utilizada), y el resultado será el siguiente:

```
MDAgMDIgM2YgNWEgNDggOTcgMjE=
```

Es exactamente la misma información, pero codificada de otra manera.

Otro ejemplo podría ser enviar como parámetro de una URL un nombre al siguiente endpoint

```
http://server-saludos.com.ar/saludar?nombre=
```

Si el nombre a enviar fuera "Rubén Random", habría caracteres que no son válidos para una url (como el espacio y el acento). Entonces se puede codificar de la siguiente manera:

```
http://server-saludos.com.ar/saludar?nombre=Rub%C3%A9n%20Random
```

Nuevamente, la información es la misma, únicamente cambia su codificación.

Encriptación

Es el proceso de convertir la información, para que no pueda ser accedida por quién no conoce como realizar el proceso inverso.

Si se quisiera encriptar el nombre del ejemplo anterior, una manera de hacerlo resultaría en lo siguiente:

```
Texto sin Encriptar: Rubén Random
Luego de Encriptar : b73eb3b98c58a1a990780002e073dbd2
```

A diferencia de la codificación, la única manera de recuperar el texto original es conocer el algoritmo y los secretos necesarios para desencriptar los datos.

Sin entrar en demasiado detalle (escapa completamente al alcance de este apunte), para realizar el proceso de encriptación / desencriptación se necesita, además del algoritmo a emplear, una llave (uno de los valores que determinarán la salida del algoritmo de encriptación). Si la misma llave que se utiliza para encriptar es la que se utiliza para desencriptar, se dice que el algoritmo de encriptación es simétrico, por ejemplo: AES, 3DES, etc.; si, por el contrario, la llave que se utiliza para desencriptar es distinta que la que se utilizó para encriptar, se dice que el algoritmo de encriptación es asimétrico, por ejemplo: RSA, EC, etc.

Firma digital

Una firma digital, en este contexto, el resultado de una serie de operaciones criptográficas que, cuando se agrega a un cierto mensaje, permite validar que el mismo no ha sido modificado y que ha sido emitido por quien dice haberlo emitido.

A modo de ejemplo, si un alumno pidiera un certificado de notas a la Universidad y la misma emitiera un PDF firmado digitalmente, el contenido de ese archivo tendría al documento y unos bytes adicionales que corresponden a la firma. Si ese certificado es entregado, por ejemplo, a una entidad gubernamental para obtener una beca, esta entidad podría analizar el contenido del documento, junto con la firma y, mediante una serie de operaciones criptográficas tener la certeza de que el certificado lo emitió la Universidad y que no ha sido adulterado.

Nuevamente, los algoritmos y detalles aquí son demasiado amplios como para incluírlos en este apunte, pero entender el concepto de firma digital servirá para entender mejor la sección sobre tokens de acceso.

HTTPS

HTTPs (HTTP Secure) hace referencia al protocolo *HTTP* pero transmitido de manera encriptada utilizando el protoclo TLS. Si bien no se explicará en este apunte el detalle de funcionamiento, entender que HTTPs se transmite de manera encriptada, permite comprender algunas advertencias que se encontrarán en las siguientes secciones del apunte.

Profundización de conceptos

Profundizando Autenticación

Como se explicó anteriormente, en un proceso de autenticación el usuario debe demostrar ser quien dice ser. En este proceso hay dos partes involucradas: el usuario que quiere autenticarse y el backend que debe decidir si las pruebas presentadas por el usuario son suficientes para considerarlo auténtico. Para entender la importancia de este proceso, basta pensar en lo que pasaría si, por ejemplo, la aplicación de un banco permitirá acceder a las cuentas de sus clientes sin validar que quien quiere acceder sea quien dice ser; cualquiera podría realizar operaciones con dinero que no es suyo.

Debe aclararse, que el término *usuario* (en el contexto de este apunte) no hace referencia, necesariamente, a un ser humano; podría tratarse, por ejemplo, de otro sistema que necesite acceder a los servicios y recursos del backend.

Entendiendo que el backend necesitará pruebas de la identidad de su usuario, es válido realizarse la siguiente pregunta: ¿Qué tipo de pruebas podría presentar un cliente?. No hay una única manera de autenticar a un usuario, pero en general estas pruebas se clasifican en:

- *Algo que el usuario conoce*: Por ejemplo, credenciales (usuario y password), OTPs, etc.
- *Algo que el usuario posee*: Por ejemplo, un dispositivo de seguridad, una smart-card, etc.
- *Algo que el usuario es*: Por ejemplo, datos biométricos como una huella digital.

Por otro lado, es razonable considerar de qué manera se presentan estas pruebas ante el backend. A continuación se explora el funcionamiento de uno de los mecanismos de autenticación existentes.

HTTP Basic

Un mecanismo de autenticación con gran popularidad es **HTTP Basic** que consiste en transmitir las credenciales del usuario (usuario + password) en el header de una petición HTTP. Este header se conforma de la siguiente manera:

```
Authorization: Basic
YWRtaW5pc3RyYWRvcjpkZWJvX2NhbWJpYXJfZXN0ZV9wYXNzMTIzNA==
```

Donde el nombre del header es *Authorization* y su valor es la cadena *Basic* seguida de un espacio y de la representación en Base64 de: *<usuario>*:*<password>*.

De esta manera, cuando el backend recibe la petición del usuario que desea autenticarse, puede leer el contenido del header *Authorization*, determinar que la autenticación es *HTTP Basic* y disponer del usuario y password para revisar si los mismos coinciden con sus registros; si el usuario y el password coinciden con los que el backend tiene almacenados, la autenticación será exitosa; caso contrario la autenticación fallará.

Lo primero que debe aclararse es que, más allá de su apariencia, la codificación en Base64 **no** es encriptación. Significa que cualquiera que acceda a la misma puede recuperar el usuario y el password sin problemas. Este tipo de transferencia de credenciales debería realizarse únicamente sobre conexiones encriptadas (Por ejemplo, utilizando HTTPS).

Debe aclararse, también, que bajo este esquema el usuario y el password deben transmitirse cada vez que el usuario necesite invocar un endpoint. Enviar el usuario y el password con cada petición significa una mayor exposición de las credenciales a los posibles atacantes.

Autenticación basada en Tokens

Se indicó que uno de los inconvenientes de la autenticación HTTP Basic es que deben enviarse las credenciales en todas las invocaciones a los endpoints. Una manera de superar este inconveniente es la utilización de *Tokens*. En este esquema las credenciales también deben enviarse al backend, pero una única vez y para ser intercambiadas por un token de acceso que será el que el usuario presentará en cada petición subsiguiente.

Sabiendo que, bajo la autenticación basada en tokens, lo que el usuario debe transmitir al backend en cada petición a un endpoint es el token obtenido (y ya no sus credenciales), la pregunta es: ¿Cómo se transmite este token?. La forma estándar para realizar esta transferencia es mediante el esquema Bearer token, que consiste en enviar un header *Authorization* y cuyo valor es la palabra *Bearer* seguida por el token, como se ejemplifica a continuación:

```
Authorization: Bearer
aICurL153yvhDZ8IICX6RM2m3HVgCFa1k3plvepApLA2vAkHDOjRJSd1Q4qYQshV
```

El token generado por el backend podría ser una cadena aleatoria (ante un login exitoso, generar y almacenar este token para luego compararlo con el que presenta el usuario en cada petición), pero esto tiene sus desventajas. Una desventaja que presenta generar un token aleatorio es que, de alguna manera, el token debe almacenarse en el backend para luego poder compararlo contra lo que envía el usuario cada vez que invoca a un endpoint.

Almacenar el token en el backend implica que cada petición tendrá que pagar el costo de esta búsqueda y su verificación (además de otros inconvenientes que se presentan si, por ejemplo, el backend fuera un sistema distribuido o hubiera varias instancias del mismo ejecutándose). Una alternativa es generar tokens que no sean cadenas aleatorias, sino que transporten consigo información suficientes para poder comprobarlos.

JSON Web Token (JWT)

De acuerdo a su especificación en la RFC-7519, un Token JWT es un formato compacto y *URL-safe* para representar un conjunto de *claims* (afirmaciones), de manera de poder transmitirse entre dos partes. Este

conjunto de *claims* se expresa como un objeto JSON, siendo cada *claim* un atributo de este objeto.

> *URL-safe* significa que podría transmitirse en una URL (por ejemplo como un query param)

Si bien la especificación de JWT no obliga al uso de un determinado conjunto de *claims*, sí existe un registro de algunos nombres de *claims* que pueden tomarse como base para facilitar la interoperabilidad de las soluciones basadas en JWT. Ellos son:

- **iss** (Issuer): Indica quién fue el emisor del *claim*
- **sub** (Subject): Indica quién es el *sujeto* sobre el cuál se están haciendo las afirmaciones (*claims*) en el token.
- **aud** (Audience): Indica cuál es el receptor para el cuál el token fue generado.
- **exp** (Expiration Time): Indica en qué momento expira (deja de ser válido) el token
- **nbf** (Not Before): Indica que el token no debe ser aceptado antes del momento especificado en este *claim*.
- **iat** (Issued At): Indica el momento de emisión del token.
- **jti** (JWT ID): Representa un identificador único para el token.

Los *claims* pueden tener cualquier nombre (mientras sean únicos en un token) y se pueden utilizar para agregar cualquier otra información que el backend necesite, por ejemplo: Detalles personales del usuario o los roles que tiene asignados en el backend (cliente, administrador, etc.).

A modo de ejemplo, si el dueño del token fuera **user123**, el token hubiera sido emitido el 14 de Octubre de 2023 a las 19:58:24 GMT y el token expirara el 14 de Octubre de 2023 a las 20:58:24 GMT, el siguiente objeto describiría esas tres afirmaciones (*claims*):

```
{
    "sub": "user123",
    "iat": 1697313504,
    "exp": 1697317104
}
```

> Los timestamps de emisión y expiración del token están expresados en segundos desde el 1/1/1970 00:00:00 GMT. Se puede visitar el sitio https://www.epochconverter.com/ si se desea convertir los segundos a Fecha/Hora y viceversa

En el contexto de este apunte y, en línea con lo expresado hasta aquí, será el usuario quien envíe este token al backend para demostrar que se autenticó correctamente y que puede acceder al endpoint deseado, mientras que será responsabilidad del server verificar que el token es válido. El problema es que, si el token únicamente contuviera el listado de *claims*, cualquier usuario podría armar su propio token y enviarlo al backend, y este último no tendría las herramientas para determinar su autenticidad. Por este motivo, no se transmiten únicamente los *claims*, sino que los mismos deben estar, o bien firmados/protegidos por MAC (especificación JWS), o bien encriptados (especificación JWE). En este apunte, al hablar de JWT se estará haciendo referencia a los primeros.

JOSE: Javascript Object Signing and Encryption es un conjunto de estándares relacionados a la firma y encriptación de datos representados en objetos JSON. Este conjunto de especificaciones incluye a JWT,

JWS (JSON Web Signing) y JWE (JSON Web Encryption), así como a JWA y JWK (Para definir las llaves y algoritmos utilizados).

En base a la explicación anterior, no alcanzará solamente con un conjunto de *claims* para definir un token JWT sino que se agregarán dos secciones más, quedando el token conformado por:

- **Encabezado (JOSE Header)**: Contiene detalles al respecto de la firma digital (o MAC) aplicado en el Token. De los atributos contenidos en este encabezado, en este apunte mencionaremos a los siguientes:
  - **alg**: Indica qué algoritmo se utilizó para la firma/MAC
  - **kid**: Indica el identificador de la clave que el backend utilizó para generar la firma/MAC
- **Contenido (*Claims*)**: Los claims descriptos hasta el momento
- **Firma/MAC (Signature)**: Los bytes que representan a la firma/MAC aplicada sobre el encabezado y los claims.

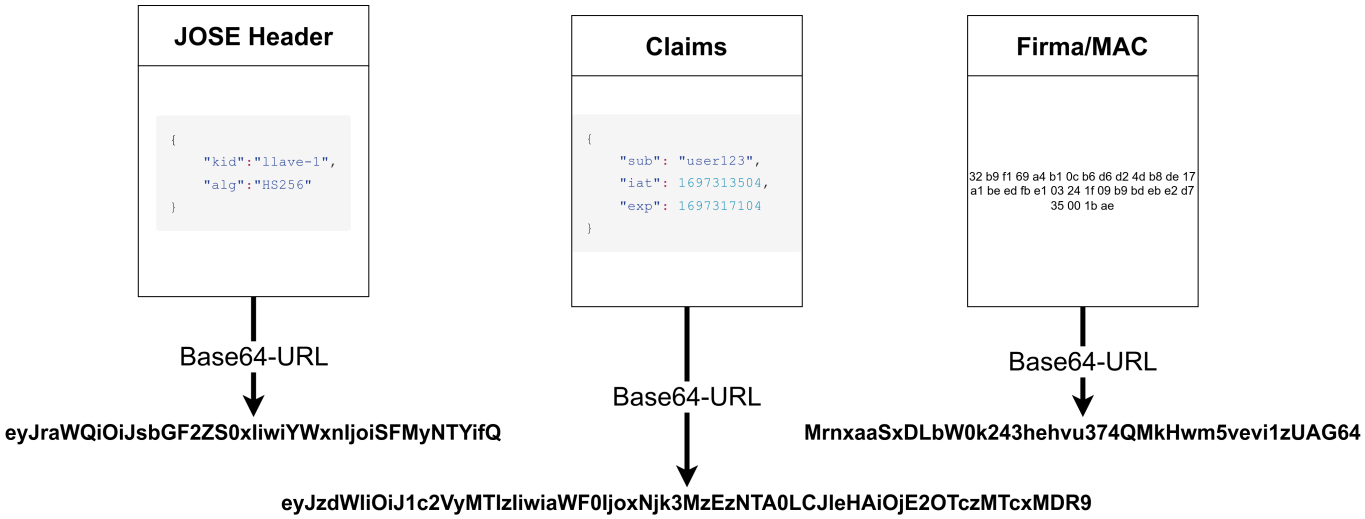
Continuando con el ejemplo anterior, si se decide utilizar HMAC con una función de Hash SHA-2/256, con una llave que en el backend recibirá el nombre de "*llave-1*", el header quedará conformado de la siguiente manera:

```
{
    "kid":"llave-1",
    "alg":"HS256"
}
```

Y al aplicar este algoritmo sobre el header y los *claims*, a partir de la llave que en nuestro ejemplo se llama "llave-1", la firma resultante será:

```
32 b9 f1 69 a4 b1 0c b6 d6 d2 4d b8 de 17 a1 be ed fb e1 03 24 1f 09 b9 bd
eb e2 d7 35 00 1b ae
// A modo de comentario, en el backend llave-1 (Expresada como String UTF-
8) es: SUPER_SECRETO_INDESCIFRABLE_BACKEND_2023
```

Con las tres secciones listas, ahora el token está completo. Pero falta un detalle adicional: Tanto el header como los *claims* están expresados en JSON, además, la firma está expresada como una serie de bytes. Los tokens JWT están pensados para que puedan ser transferidos en headers HTTP o, incluso, URLs, entonces el paso final para codificar el token de una manera que pueda ser transferido en estas peticiones consiste en que cada una de las secciones será codificado en Base64-URL y cada una de estas cadenas separadas por un caracter "." (A esto se lo llama serialización en forma compacta).



A continuación se puede ver el token del ejemplo en su forma compacta (Notar los puntos que separan las tres secciones):

```
eyJraWQiOiJsbGF2ZS0xIiwiYWxnIjoiSFMyNTYifQ.eyJzdWIiOiJ1c2VyMTIzIiwiaWF0Ijo
xNjk3MzEzNTA0LCJleHAiOjE2OTczMTcxMDR9.MrnxaaSxDLbW0k243hehvu374QMkHwm5vevi
1zUAG64
```

A partir de haber obtenido este token, cuando el usuario desee acceder a un endpoint del backend, enviará una petición con el siguiente header:

```
Authorization: Bearer
eyJraWQiOiJsbGF2ZS0xIiwiYWxnIjoiSFMyNTYifQ.eyJzdWIiOiJ1c2VyMTIzIiwiaWF0Ijo
xNjk3MzEzNTA0LCJleHAiOjE2OTczMTcxMDR9.MrnxaaSxDLbW0k243hehvu374QMkHwm5vevi
1zUAG64
```

Como ahora el token contiene una firma digital, cuando el usuario lo envíe al backend para acceder a un endpoint, el backend podrá controlar esta firma y saber que es un token emitido por él (por el backend) y que su contenido no fue alterado. Con esto se logra que el backend no necesite almacenar los tokens, pero igualmente pueda validar que son auténticos.

El proceso que hará el backend ante cada petición que contenga un token JWT será inspeccionar el header, validar el contenido del token de acuerdo a la firma y, si el token es válido, extraer los *claims* que necesite, por ejemplo, para saber si el usuario (que ya está autenticado) tiene permisos sobre los recursos a los que está queriendo acceder.

El proceso de generar y validar un token JWT no es tan sencillo como parece, hay múltiples detalles a tener en cuenta, por ese motivo no se recomienda realizar implementaciones "artesanales" de este proceso. Afortunadamente, existe una gran cantidad de librerías que permiten trabajar con este tipo de tokens.

Código de ejemplo

Para este apunte se seleccionó la librería **jose4j**, cuyas coordenadas son:

```
        <dependency>
            <groupId>org.bitbucket.b_c</groupId>
            <artifactId>jose4j</artifactId>
            <version>0.9.3</version>
        </dependency>
```

El siguiente código muestra un sencillo ejemplo de cómo se puede utilizar esta librería para generar y para validar un token.

```
import org.jose4j.jwa.AlgorithmConstraints;
import org.jose4j.jws.AlgorithmIdentifiers;
import org.jose4j.jws.JsonWebSignature;
import org.jose4j.jwt.*;
import org.jose4j.jwt.consumer.*;
import org.jose4j.keys.HmacKey;
import org.jose4j.lang.JoseException;

import java.nio.charset.StandardCharsets;
import java.util.Map;


public class ManejadorTokens {

    private static final String NOMBRE_LLAVE = "llave-1";
    private static final String ALGORITMO =
AlgorithmIdentifiers.HMAC_SHA256;
    private static final String SECRETO =
"SUPER_SECRETO_INDESCIFRABLE_BACKEND_2023";
    private static final HmacKey hmacKey = new
HmacKey(SECRETO.getBytes(StandardCharsets.UTF_8));

    public String crearToken(String usuario, long segundosValidez) throws
JoseException {

        // Fecha/hora actual
        var issuedAt = NumericDate.now();
        // Copia de la Fecha/Hora actual
        var expiresAt = NumericDate.fromSeconds(issuedAt.getValue());
        // La expiración será segundosValidez luego de la emisión
        expiresAt.addSeconds(segundosValidez);

        // Creación de los claims. Aquí se pueden agregar claims
adicionales.
        JwtClaims claims = new JwtClaims();
        claims.setSubject(usuario);
        claims.setIssuedAt(issuedAt);
        claims.setExpirationTime(expiresAt);

        // El token va a incluír una firma
        JsonWebSignature jws = new JsonWebSignature();
```

```
        // La firma se realiza a partir de una llave y aplicando un cierto
algoritmo
        jws.setKeyIdHeaderValue(NOMBRE_LLAVE);
        jws.setKey(hmacKey);
        jws.setAlgorithmHeaderValue(ALGORITMO);

        // Hay que setear los claims creados anteriormente
        jws.setPayload(claims.toJson());

        // Finalmente, hay que recordar que para que sea transmitido debe
serializarse en su forma compacta
        return jws.getCompactSerialization();
    }

    public Map<String, Object> verificarToken(String token) throws
InvalidJwtException {

        // jose4j utiliza la clase JwtConsumer para realizar la validación
        JwtConsumer jwtConsumer = new JwtConsumerBuilder()
                .setRequireExpirationTime() // Se requiere que tenga el
claim "exp"
                .setRequireSubject() // Se requiere que tenga el claim
"sub"
                .setVerificationKey(hmacKey) // Necesita la clave para
poder verificar la firma
                .setJwsAlgorithmConstraints(
                        AlgorithmConstraints.ConstraintType.PERMIT,
ALGORITMO
                ) // Que controle que el header contenga al algoritmo
esperado
                .build(); // create the JwtConsumer instance

        // Una vez construído el consumidor, se lo procesa. Ante un
problema al validarlo,
        // se arroja la excepción InvalidJwtException. La excepción
contiene información al
        // respecto de qué validación falló (si la firma o si expiró,
etc.)
        JwtClaims claims = jwtConsumer.processToClaims(token);

        // Retorno los claims como una cadena JSON
        return claims.getClaimsMap();
    }
}
```

Claramente, el secreto, el tipo de algoritmo y otros detalles no estarían presentes en el código fuente de una aplicación real. También debe notarse que el método *crearToken()* únicamente agrega tres *claims*, para agregar *claims* adicionales, sencillamente habría que utilizar el método

```
public void setClaim(String claimName, Object value);
```

de la clase *JwtClaims*.

Nota: En el *Apéndice 1* de este apunte se encuentran las instrucciones para ver en la práctica estos conceptos.

Profundizando Autorización

En la sección anterior de este apunte se trató el tema de la *Autenticación*, cuyo objetivo era identificar al usuario realizando la petición. Una vez que se sabe que el usuario es quien dice ser, se debe determinar a qué recursos tiene acceso; en otras palabras: qué puede hacer ese cliente en el backend, a este proceso se lo llama *Autorización*. A modo de ejemplo, si un usuario inicia sesión en el sitio de una cadena de electrodomésticos podrá listar los artículos disponibles y realizar compras pero no podrá cambiar el precio de los artículos; sin embargo, existirán otros usuarios con un rol más privilegiado que sean capaces de realizar estos cambios.

Los esquemas de autenticación estudiados hasta aquí permitían verificar que el usuario de los endpoints estuviera *autenticado*, pero no tenían información al respecto de que privilegios tenían estos usuarios respecto a los recursos que administran los endpoints. En definitiva, falta información para realizar el proceso de *autorización*, esto se analizará en las siguientes secciones.

OAuth2

Para comunicarse con un backend y poder acceder a los recursos que el mismo mantiene, el mismo proporciona una API y, aunque no es obligatorio que sea así, lo más frecuente al día de hoy es que esta API se acceda mediante peticiones HTTP. Es muy poco probable que el usuario final (una persona) vaya a interactuar de esta manera con un backend (basta imaginar lo limitado e impráctico que sería acceder a un homebaking si esta fuera la forma de hacerlo), entonces será necesario que el usuario deba utilizar alguna aplicación para este fin (por ejemplo la app móvil del banco) y será necesario, también, que de alguna manera el usuario le indique al backend que esta aplicación puede acceder a la información de sus cuentas, tarjetas, etc.

La aplicación que el usuario utilice para acceder a los recursos del backend, que puede ser una aplicación web, una aplicación móvil, una aplicación de escritorio, etc., podría pedirle las credenciales al usuario y utilizarlas como si fueran propias para acceder al backend, pero hay algunos inconvenientes con este enfoque, por ejemplo:

- La aplicación, a partir de autenticarse como si fuera el usuario, tendría acceso a cualquier actividad que el usuario tenga permitida en el backend. No habría una forma de limitar que es lo que la aplicación puede hacer y qué es lo que no.
- A menos que las credenciales cambien (lo que no sucede tan frecuentemente), la aplicación tendrá acceso permanente a los recursos del usuario.
- El usuario está entregando sus credenciales y debe confiar en que la aplicación se comporte correctamente y haga uso debido de las mismas.

Entonces se presenta un escenario donde una tercera parte (en este caso la aplicación) necesita que se le autorice el acceso a recursos que no le son propios (que son del usuario) y que esto se realice de una

manera acotada (respecto a lo que puede hacer y respecto al espacio de tiempo durante el cuál lo puede hacer).

OAuth2 es un framework de autorización pensado resolver el problema de accesos delegados que se explicó previamente. En la terminología de OAuth2, el listado de las autorizaciones entregadas al cliente (lo que puede hacer) se denomina *scope*.

> La palabra *framework* en este caso no hace referencia a un framework de software (como el caso de Spring), sino a un conjunto de definiciones y procedimientos que deben implementarse.

Lo primero que debe entenderse, al respecto de OAuth2, son los roles que define. Estos son cuatro y, en palabras de la *RFC-6749*, son:

- **Resource Owner**: Una entidad capaz de otorgar acceso a un recurso protegido. Cuando el propietario del recurso es una persona, se le denomina usuario final. En el ejemplo de un homebanking, es la persona dueña de las cuentas, de la información de sus tarjetas, etc.
- **Resource Server**: El servidor que aloja los recursos protegidos, capaz de aceptar y responder a solicitudes de recursos protegidos utilizando tokens de acceso. En el ejemplo anterior, serían los servicios del backend del banco que permiten ver los saldos, realizar transferencias, etc.
- **Client**: Una aplicación que realiza solicitudes de recursos protegidos en nombre del propietario del recurso y con su autorización. El término "cliente" no implica ninguna característica de implementación particular. En el ejemplo anterior, sería la aplicación móvil (o la aplicación web) del banco y va a necesitar conseguir un token de acceso para poder invocar a los endpoints del backend.
- **Authorization Server**: El servidor que emite tokens de acceso al cliente después de realizar con éxito la autenticación del propietario del recurso y obtener su autorización. En OAuth2, el emisor de los tokens tiene un rol específico como servidor de autorización.

Nota: Se debe destacar que la especificación de OAuth2 no indica que formato deben tener los tokens. En otras palabras, no es obligatorio que los tokens sean JWT, aunque muchas implementaciones de servidores de autorización los generan bajo este estándar.

Generación de los Tokens

Habiendo comprendido cuáles son los roles definidos por OAuth2, se debe comprender que no todos los clientes son iguales: No es lo mismo una aplicación cliente que opera del lado del servidor (como podrían ser aplicaciones JSP, SpringMVC, etc.) y que pueden almacenar secretos de forma segura, que aplicaciones web implementadas como SPAs o aplicaciones de escritorio; tampoco hay que olvidarse de clientes que, siendo un backend, deben comunicarse con otro (un ejemplo de esto podría ser un backend que deba comunicarse con un autorizador de tarjetas de crédito o un backend que emita facturas utilizando los servicios de AFIP). Entonces, existen distintos flujos (*grant types*) para obtener los tokens de acceso por parte de estos clientes. Ellos son:

- **Authorization Code:** Requiere que el *User Agent* (Generalmente un browser) que utiliza el Resource Owner (El usuario final) tenga la capacidad de reaccionar ante redirecciones, entonces el flujo procede como sigue:
    1. El Cliente (la aplicación) redirecciona al Resource Owner (el usuario final) al Authorization Server, indicando cuál es el *scope* solicitado.

    2. El Authorization Server autentica al Resource Owner (generalmente mediante un formulario de login) y solicita la autorización del para que el Cliente acceda a los recursos protegidos.
    3. El Authorization Server redirecciona al Resource Owner hacia el Cliente, incluyendo un código de autorización.
    4. El Cliente realiza una petición al endpoint de tokens del Authorization Server para cambiarlo por el token (que es el que luego utilizará para cada petición al Resource Server).
- **Implicit:** Es una variación de *Authorization Code*, donde no hay un código de autorización intermedio (directamente se entrega el token), y no se autentica al usuario en el Authorization Server. No se recomienda la utilización de este flujo.
- **Resource Owner Password Credentials:** Únicamente se recomienda si el Resource Owner confía en el Cliente (ya que involucra compartir credenciales, algo que se quiere evitar, justamente, utilizando OAuth2). El flujo procede de la siguiente manera:
    1. El Resource Owner envía sus credenciales al Cliente.
    2. El Cliente realiza una petición al Authorization Server enviando el *scope* deseado, las credenciales del Resource Owner y su Client ID + Client Secret (en el caso más común el Cliente debe estar registrado en el Authorization Server).
    3. El Authorization Server, luego de la autenticación, responde al Cliente con el token de acceso.
- **Client Credentials:** Pensado para cuando el Cliente requiere la autenticación pero a su nombre (no a nombre del Resource Owner). Se aplica, por ejemplo, para comunicación entre backends.

Validación de los Tokens

Los endpoints del backend (*Resource Owners*, en la terminología OAuth2), deben responder, únicamente, a peticiones autorizadas; por este motivo el Cliente que realiza las peticiones va a enviar el token que obtuvo previamente en cada petición que realice y el backend deberá validar esos tokens. Para realizar esta validación hay dos alternativas:

1. Realizar una invocación al Authorization Server, enviando el token, y será el Authorization Server el que realice las validaciones necesarias y le responda al Resource Server si el token es válido o no (entre otros detalles). La principal desventaja de este enfoque es que se debe pagar el costo de la invocación al Authorization Server cada vez que el Resource Server responda a una petición.
2. Nuevamente, no es obligatorio, pero si la implementación del Authorization Server utiliza tokens autocontenidos (como JWT), el mismo Resource Server puede validar la firma del token (o desencriptar el token si fuera el caso) y obtener directamente los grants contenidos en él.

Open ID Connect

Open ID Connect, es una extensión de OAuth2 que cubre (además) la capa de autorización que no está presente en OAuth2.

Entre otras modificaciones, agrega funcionalidades no disponibles (o no estandarizadas). Por ejemplo:

- En OAuth2, el token de acceso es opaco para el Cliente. OIDC agrega un nuevo tipo de token (*ID token*) y requiere que el mismo sea JWT, de manera que el Cliente puede obtener, por si mismo, detalles del cliente autorizado. Por otro lado, OIDC requiere que el Authorization Server provea un endpoint *User Info* para obtener detalles respecto del usuario, el tipo de autenticación, etc.
- Se permite al Cliente indicar la forma de autenticación que debería realizar el Usuario.
- El Cliente puede ser notificado respecto al estado de sesión del usuario.

OIDC también define roles participantes, en algunos casos equivalentes a los de OAuth2. Los mismos son:

- **End User:** Un mejor nombra para el usuario final. Es el equivalente al *Resource Owner* de *OAuth2*.
- **Relaying Party:** Es la aplicación que desea autenticar al usuario final.
- **OpenID Provider:** Es el servidor que autentica al usuario.

Otro término que no varía en relación a *OAuth2* es el de *grant type*, en el caso de OIDC se llaman *flows* (lo que representa un término menos confuso). Los dos flujos principales de OIDC son:

- **Authorization Code Flow:** Es el flujo ya definido en *OAuth2*, pero al intercambiar el código de acceso por el token de acceso, se agrega el *ID token* explicado previamente.
- **Hybrid flow**: A diferencia del caso anterior, el *ID token* está disponible junto con el código de acceso (se obtiene previo al paso de intercambio del código de acceso por los tokens).

El Servidor de Autorización

Existen múltiples herramientas y frameworks que se pueden utilizar para implementar un Servidor de Autorización. Desde proyectos de Spring Framework y software Open Source hasta servicios de autorización listos para utilizar.

En el caso de esta materia, las prácticas se realizarán utilizando una instancia de **KeyCloak**. KeyCloak es un proyecto Open Source que sirve para la gestión de Identidad y Acceso, básicamente permite la gestión de los procesos de *Autenticación* y *Autorización*. Se trata de una herramienta muy completa, con gran flexibilidad de configuración y que soporta, entre otros estándares, *OAuth2* y *Open ID Connect*. En el caso de *OAuth2*, los tokens de acceso generado son tokens JWT, permitiendo una integración más sencilla.

En los siguientes apuntes se tratará con más detalle la interacción de los clientes y el backend con este servidor.

Apéndice 1

En este apéndice se trabajará con muy pequeño servidor que tiene dos endpoints:

1. http://localhost:8080/auth que permite autenticar a un usuario con el backend y generar un token que se podrá utilizar para llamar a los demás endpoints.
2. http://localhost:8080/api/v1/random que representa a un endpoint que está protegido y debe ser invocado presentando el token generado con el endpoint /auth. Este endpoint lo que hace es generar un valor entero aleatorio y retornarlo. Además, agrega en la respuesta los claims presentes del token a modo de demostración.

Para ver en acción la generación de un token JWT y su validación siga los siguientes pasos:

1. Descargue del repositorio el proyecto **JWTDemo** y ejecútelo. Esto levantará un servidor en *http://localhost:8080/*. Esta ejecución se puede hacer abriendo el proyecto desde un IDE o con el comando

   ```
   mvn spring-boot:run
   ```

2. Abra **Postman** y genere una nueva colección.

3. Dentro de la colección creada en el paso anterior, cree una nueva petición.
    1. Cambie el tipo de petición a POST y la ruta a http://localhost:8080/auth
    2. En la pestaña *Authorization* seleccione el tipo *Basic Auth*
    3. En el campo *Username* ingrese el valor *usuario123*
    4. En el campo *Password* ingrese el valor *password123*
    5. Presione el botón *Send*
    6. Deberá recibir del endpoint de autorización una respuesta similar a la siguiente:

```
    {
        "token":
"eyJraWQiOiJsbGF2ZS0xIiwiYWxnIjoiSFMyNTYifQ.eyJzdWIiOiJ1c3VhcmlvM
TIzIiwiaWF0IjoxNjk2ODEzOTk0LCJleHAiOjE2OTY4MTQyOTR9.xmzHEF_AD7n8C
DcKHq8ISYkreuLlwm8FkoBfSPq9DgQ"
    }
```

    7. Esta respuesta contiene el token JWT generado como resultado del proceso de autenticación. Copie el valor del token (solamente el token, en este ejemplo: eyJraWQiOiJsbGF2ZS0xIiwiYWxnIjoiSFMyNTYifQ.eyJzdWIiOiJ1c3VhcmlvMTIzIiwiaWF0IjoxNjk2ODEzOTk0LCJleHAiOjE2OTY4MTQyOTR9.xmzHEF_AD7n8CDcKHq8ISYkreuLlwm8FkoBfSPq9DgQ)

4. Dentro de la colección creada en el punto 2, cree una nueva petición.
    1. Cambie el tipo de petición a GET y la ruta a http://localhost:8080/api/v1/random
    2. En la pestaña *Authorization* seleccione el tipo *Bearer Token*
    3. En el campo *Token* ingrese el valor del token copiado anteriormente
    4. Presione el botón *Send*
    5. Deberá recibir del endpoint *random* una respuesta similar a la siguiente:

```
    {
        "valorAleatorio": -7196049389396417367,
        "detallesToken": {
            "sub": "usuario123",
            "iat": 1696813994,
            "exp": 1696814294
        }
    }
```

    6. Altere el valor del campo Token en la petición y presione el botón *Send*, la respuesta del endpoint será 401 (Unauthorized)

7. Restaure el valor del campo Token al valor devuelto en el punto 3.6. Espere que pase el tiempo de duración del token (5 minutos desde la autenticación) y presione *Send* nuevamente. La respuesta del endpoint será 401 (Unauthorized), ya que se superó el tiempo de expiración del token.